2014 47th Annual IEEE/ACM International Symposium on Microarchitecture

COMP: Compiler Optimizations for Manycore Processors

Linhai Song*, Min Feng†, Nishkam Ravi‡, Yi Yang† and Srimat Chakradhar†
*Computer Sciences Department
University of Wisconsin-Madison
Madison, WI, USA
songlh@cs.wisc.edu
†Computer Systems Architecture Department
NEC Laboratories America
Princeton, NJ, USA
{mfeng,yyang,chak}@nec-labs.com
‡Cloudera Inc.
Palo Alto, CA, USA
nravi@cloudera.com

***Abstract*—Applications executing on multicore processors can now easily offload computations to manycore processors, such as Intel Xeon Phi coprocessors. However, it requires high levels of expertise and effort to tune such offloaded applications to realize high-performance execution. Previous efforts have focused on optimizing the execution of offloaded computations on manycore processors. However, we observe that the data transfer overhead between multicore and manycore processors, and the limited device memories of manycore processors often constrain the performance gains that are possible by offloading computations.**

In this paper, we present three source-to-source compiler optimizations that can significantly improve the performance of applications that offload computations to manycore processors. The first optimization automatically transforms offloaded codes to enable data streaming, which overlaps data transfer between multicore and manycore processors with computations on these processors to hide data transfer overhead. This optimization is also designed to minimize the memory usage on manycore processors, while achieving the optimal performance. The second compiler optimization re-orders computations to regularize irregular memory accesses. It enables data streaming and vectorization on manycore processors, even when the memory access patterns in the original source codes are irregular. Finally, our new shared memory mechanism provides efficient support for transferring large pointer-based data structures between hosts and manycore processors.

Our evaluation shows that the proposed compiler optimizations benefit 9 out of 12 benchmarks. Compared with simply offloading the original parallel implementations of these benchmarks, we can achieve 1.16x–52.21x speedups.

I. Introduction

A. Motivation

Although manycore processors have the ability to provide high performance, achieving such performance on them remains a challenging issue. It usually requires very high levels of expertise and effort from programmers to understand and make good use of the underlying architectures. For example, to develop high performance GPU applications, programmers need to be aware of the memory hierarchy and the warp-based thread organization, given their dominant impact on performance. Many static and runtime techniques have been developed to relieve the optimization burden from programmers developing GPU applications [23], [29], [30]. These efforts have mainly focused on optimizing the execution of offloaded computations on manycore processors. However, a recent study [20] shows that there is still a significant performance gap between compiler optimized codes and highly tuned CUDA codes.

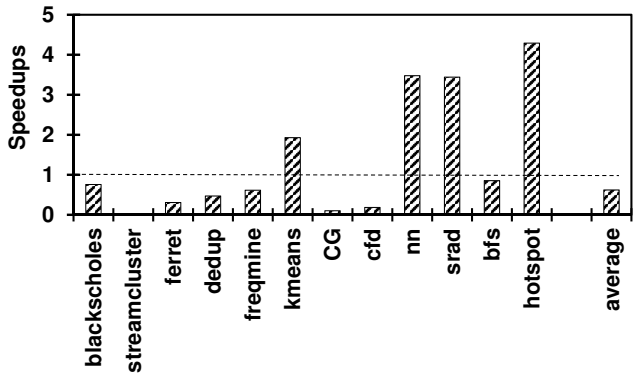

Figure 1: Speedups of OpenMP codes on a Xeon Phi coprocessor compared with a multicore CPU

Recently, Intel introduced Xeon Phi coprocessors as new members of the manycore family. They are based on the Intel Many Integrated Core Architecture (Intel MIC Architecture). Compared to Nvidia GPUs, Xeon Phi coprocessors are easier to program. They provide x86 compatibility and support many different programming models, like OpenMP [7], TBB [10], and Cilk Plus [14]. In order to port software to Xeon Phi coprocessors, developers just need to add simple pragmas into their existing codes to offload parallel loops to

1072-4451/14 $31.00 © 2014 IEEE
DOI 10.1109/MICRO.2014.30

the coprocessors.

However, our recent experience shows that achieving good performance on Xeon Phi coprocessors is still not a straightforward task for programmers. Figure 1 compares the speedups of a set of OpenMP benchmarks on a multicore CPU and a Xeon Phi coprocessor. To run these benchmarks on the Xeon Phi coprocessor, we add pragmas to offload the parallel loops. The parallel performance on the CPU is measured using 4–6 threads, while the Xeon Phi performance is measured using 200 threads. All speedups are normalized to the CPU versions. The experimental setting will be explained in more detail in Section VI. Although these programs are intrinsically parallel with minimal communication and expected to have high performance on manycore processors, Figure 1 shows that the Xeon Phi coprocessor performs poorly for 8 out of 12 benchmarks compared to the CPU. For some of these benchmarks, the Xeon Phi performance is even worse than the sequential performance on the CPU. Simply offloading the parallel loops does not give performance on the Xeon Phi coprocessor.

B. Problems

In this paper, we show that traditional optimizations on offloaded computations are not enough to guarantee better performance on manycore processors. After careful investigation, we identify three performance bottlenecks:

Firstly, the data transfer overhead between multicore and manycore processors is huge. Data need to be transferred through PCIe bus to manycore processors before offloaded computations are conducted. Our experimental results show that data transfer is expensive, and for some benchmarks, it even takes a longer time than computations on the coprocessor. In order to achieve better performance, we need to find a way to reduce the data transfer overhead.

Secondly, irregular memory accesses hurt cache locality, disable vectorization, and usually incur poor performance on manycore processors. Compared with multicore processors, manycore processors have less per-core memory bandwidth and support longer (512 bits) SIMD operations. Static or dynamic techniques are desired to regularize irregular memory accesses.

Finally, the current data transfer mechanism for large pointer-based data structures does not work efficiently. Intel provides MYO [28] shared memory abstraction to transfer complex data structures between multicore and manycore processors. Our experience shows that MYO is very slow, due to the small data transfer granularity and the large number of page fault handling. A new data transfer mechanism is needed for large pointer-based data structures.

Currently, it relies on programmers tuning the offloaded codes to overcome these performance bottlenecks. However, tuning codes for manycore processors requires high levels of expertise and effort, and is a challenging task for programmers. To reduce the data transfer overhead, programmers need to rewrite loops in a pipelined style and use asynchronous communications to overlap data transfers with computations. They also need to balance the memory usage and performance on manycore processors and choose proper synchronization granularity to optimize the kernel launching overhead. Handling irregular memory accesses requires programmers to understand the memory access patterns. Each offloaded loop may require a specific optimization to achieve the optimal performance. For pointer-based data structures, programmers can use linearization to efficiently transfer these data structures. However, reconstructing the links (i.e., pointers) between objects on manycore processors not only takes efforts, but also incurs extra overhead.

C. Our Proposal

This paper proposes three compiler optimizations to address identified performance issues. Compared with manually tuning programs, compiler optimizations mitigate burden from programmers and allow them to focus on parallelism extraction. Although these optimizations are presented in the context of Intel Xeon Phi coprocessors, we believe that these techniques can also be applied to other emerging manycore processors, such as the Tilera Tile-Gx processors.

The first optimization, *data streaming*, is designed to reduce the overhead of transferring data between CPUs and coprocessors. The optimization automatically transforms offloaded codes to stream data to and from coprocessors, which overlaps data transfers with computations to hide the data transfer overhead. To enable streaming data, we develop compiler techniques to solve two practical issues: (1) how do we minimize the memory usage on manycore processors, while achieving the optimal performance, and (2) how do we reduce the kernel launching overhead caused by *data streaming*.

The second optimization, *regularization*, is designed to handle loops with irregular memory accesses. The optimization identifies irregular memory access patterns in a loop and re-arranges the order of computations to regularize memory accesses. It enables *data streaming* and *vectorization* for manycore processors in the presence of irregular memory accesses. It also improves cache locality and relieves the memory bandwidth bottleneck.

Finally, we develop a new shared memory mechanism to support the efficient transfer of large pointer-based data structures between CPUs and coprocessors. Our memory allocation scheme is designed to optimize the memory usage on manycore processors. An augmented design of pointers is introduced for fast translating pointers between their CPU and coprocessor memory addresses.

We evaluate the proposed optimizations using 12 benchmarks from PARSEC [13], Phoenix [26], NAS [4], and Rodinia [17]. The experimental results show that our optimizations improve the performance for 9 out of 12 benchmarks.

Overall, our optimizations improve the MIC performance by 1.16x–52.21x.

Our paper makes the following contributions:

A thorough performance study for Intel Xeon Phi coprocessors. We study the performance of a set of OpenMP benchmarks on a machine equipped with a Xeon Phi coprocessor. The study shows that directly executing the OpenMP parallel loops on the coprocessor will either give poor performance or cause runtime errors. We identify the root causes of these issues, which are the data transfer time between the CPU and the coprocessor, irregular memory accesses, and limited shared memory spaces between the CPU and the coprocessor.

Three proposed compiler optimizations to address identified performance issues. They improve the manycore coprocessor performance and enable the execution of the computation tasks that previously cannot be executed on the coprocessor. The compiler techniques also reduce the expertise and effort required for programming manycore processors. Although the techniques are described in the context of Xeon Phi coprocessors, we believe that they can also be applied to other manycore processors.

A thorough evaluation for proposed compiler optimizations. We evaluate the three compiler optimizations using a number of benchmarks. We show that performance can be achieved by using a combination of the three optimizations.

The rest of the paper is organized as follows. Section II briefly describes the execution and programming models for Xeon Phi coprocessors. In Section III, Section IV and Section V, we present the details of the three compiler optimizations. In Section VI, we describe our evaluation for the three compiler optimizations. Section VII discusses the related works and Section VIII concludes the paper.

II. Background

In this section, we first briefly describe the Intel MIC Architecture used by Xeon Phi coprocessors, and then illustrate its execution and programming models.

A. Intel MIC Architecture

The Intel MIC Architecture is designed to enable high levels of thread and SIMD parallelism. The most recent Xeon Phi processor contains 61 cores connected through a ring bus. Each core is a modified first-generation Pentium processor that supports 4 simultaneous threads. Since one core is reserved for OS use, user applications can use up to 240 simultaneous threads in total. While the single thread performance of Intel MIC is worse than that of a modern CPU, it provides more scalable performance for parallel applications. To further increase the parallelism, Intel MIC supports 512-bit SIMD operations. Vectorization is thereby a key to achieving performance.

The memory management of Intel MIC is similar to that on CPUs. All cores in the Intel MIC Architecture share an 8 GB memory with coherence L2 cache. As a coprocessor, Intel MIC has no disk to access, and it has no swap space to switch out unused memory pages. More efficient memory usage is required to run applications with big memory footprints.

The Intel MIC Architecture is x86 compatible. It supports standard programming languages, such as Fortran and C/C++, and can run legacy CPU codes. It can also utilize existing parallelization tools for CPUs, such as OpenMP [7], OpenCL [6], and Cilk/Cilk Plus [14].

B. Offload Mode

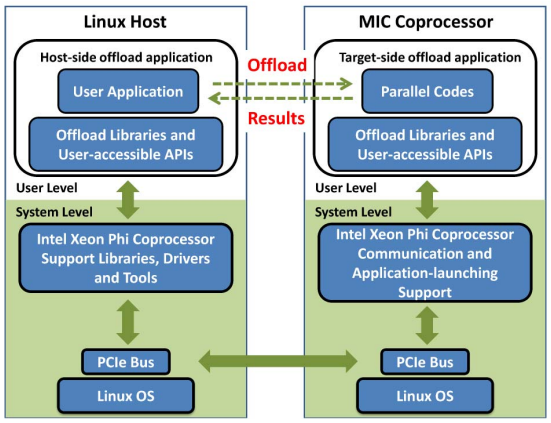

Figure 2: Intel Xeon Phi overview and the offload mode

Intel provides both native mode and offload mode for MIC coprocessors. In native mode, all codes are ported to MIC. Since an application usually has a portion of serial codes [11], and the performance of a single MIC thread is much worse than a single CPU thread, native mode is quite slow.

Offload mode allows serial codes to be executed on a modern CPU, which has a higher clock speed and a more advanced architecture, and only offloads the highly-parallel code regions of an application to a coprocessor to achieve more scalable performance, as shown in Figure 2. In offload mode, an application is always started on a CPU. Once the execution gets into a parallel code region, it copies the parallel code and input data from the CPU to a coprocessor through PCIe bus, and then executes the parallel code on the coprocessor. After the parallel code region is done, the output data are copied back from the coprocessor to the CPU, and the execution continues on the CPU.

C. Offload Programming

Intel provides LEO (Language Extension for Offload) for programming the offload mode. It is a set of high-level directives designed to improve programmer productivity. LEO is very similar to OpenACC [5]. Programmers can choose what codes to offload and need to specify the input

and output data for offloaded code regions. Sub-figure (a) in Figure 5 shows a LEO code example extracted from benchmark `blackscholes`. The offloaded code region is a loop parallelized with OpenMP pragmas. An *offload* pragma is inserted before the loop to specify the offloaded code region. The *target* clause gives where the code will be offloaded. In this case, it is the first MIC device. The *in* and *out* clauses are used to declare the input and output data for the offloaded code region.

```
int * _Cilk_shared v;
_Cilk_shared void foo() {
    for(int i = 0; i < 5; i++) {
        v[i] = i;
    }
}
int main() {
    int size = sizeof(int)*5;
    v = (int * _Cilk_shared)_Offload_shared_malloc(size);
   _Cilk_offload foo();
   return 0;
}
```

Figure 3: A LEO code example using shared memory between the CPU and the coprocessor

In addition to explicitly specifying data to be transferred, LEO also supports an implicit data transfer model with shared memory between CPUs and coprocessors. A runtime called MYO [28] is designed to automate the procedure of transferring shared data between a CPU and a coprocessor. Figure 3 shows a code example using MYO. In the example, variable v marked with *_Cilk_shared* and the data allocated using *Offload_shared_malloc* are shared between the CPU and the coprocessor. The *_Cilk_offload* clause is used to offload function *foo()* to the coprocessor. The data synchronization of variable v occurs at the boundary of the offloaded code region, according to the MYO scheme.

III. Data Streaming

This section presents the *data streaming* optimization. We first show the data transfer overhead for a set of benchmarks and explain how *data streaming* mitigates this issue. We then illustrate the compiler transformation for *data streaming* and two optimizations for further reducing the time and memory overhead.

Figure 4 shows the data transfer overhead for a number of benchmarks. In the figure, we compare data transfer time and calculation time for benchmarks `blackscholes`, `kmeans`, and `nn`. For each benchmark, data transfer time is normalized by calculation time on MIC. We can see that data transfer takes quite a long time for these benchmarks. Therefore, optimizing data transfer is critical to the performance of these benchmarks.

We propose a compiler optimization, *data streaming*, which automatically transforms offloaded codes to overlap

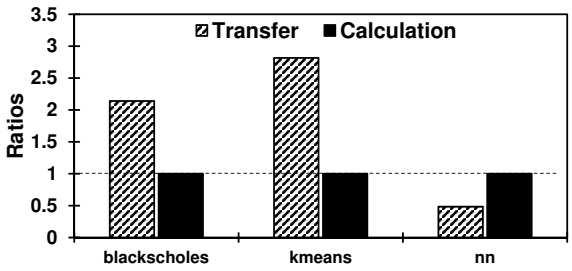

Figure 4: Data transfer overheads

data transfer with computation to hide data transfer time. Sub-figure (d) in Figure 5 shows the basic idea of *data streaming* and how it can reduce the data transfer overhead. Normally to execute a code region on a coprocessor, we start the computation after finishing transferring all the input data to the coprocessor. The total execution time is the computation time plus the data transfer time. With *data streaming*, both data transfer and computation are divided into multiple blocks and performed in a pipelined style. The *i*-th computation block, which is a subset of continuous iterations, starts right after the *i*-th data block is transferred to the coprocessor and overlaps with the data transfer of the *(i+1)*-th block. Ideally, the total execution time is then reduced to the computation time plus the data transfer time of the first block. Previously, this optimization is done by programmers who have the expertise and application knowledge to make the transformation correct and efficient. We believe that relieving programmers of this low-level optimization will help them better focus on algorithmic logic and parallelism extraction.

A. Code Transformation

The code transformation for *data streaming* is described in detail as follows. We use the loop in `blackscholes` benchmark as an example for illustration. Sub-figure (b) in Figure 5 shows the transformed loop, which overlaps the data transfer of the price arrays with the calculation of function `BlkSchlsEqEuroNoDiv()`. The transformed loop contains four sections, which perform memory allocations, data transfer for the first block, the main loop, and memory deallocation, respectively.

Legality check. Before performing the code transformation on a loop, we need to check if *data streaming* can be applied to the loop. For a loop executed in *data streaming* model, we start a computation block, when its input data are ready. Therefore, to automatically apply *data streaming*, compiler needs to understand what input data are required for each computation block. In our implementation, we apply *data streaming* only when all array indexes in a loop are in the form of $a * i + b$, where i is the loop index and a and b are constants. This allows us to easily calculate the

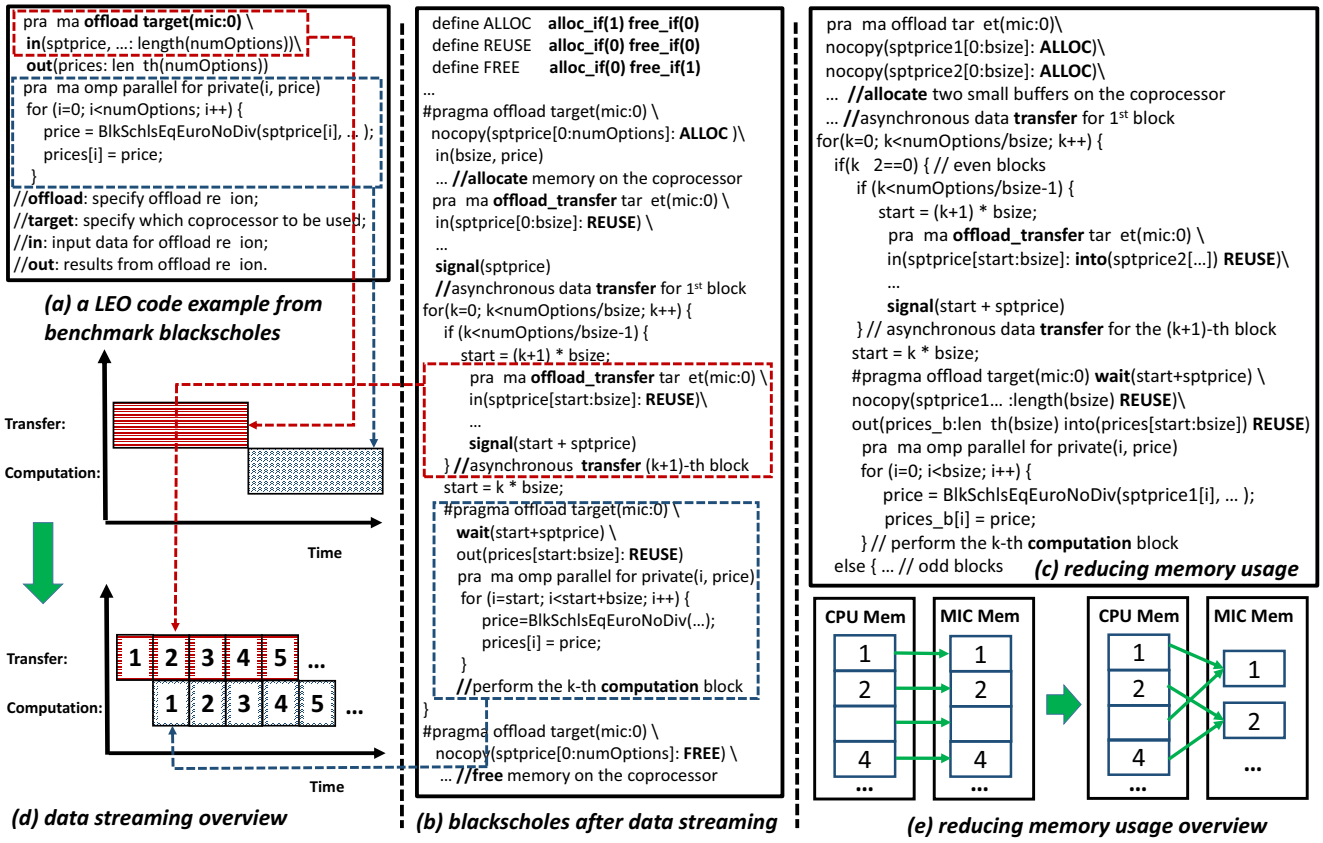

Figure 5: Blackscholes example. (a): how to offload parallel loops to the coprocessor; (b): source codes after applying the data streaming transformation; (c): source codes after applying the reducing memory usage optimization; (d): basic idea of data streaming; (e): basic idea of how to reduce memory usage

data portion that is required by a computation block. In the example of `blackscholes`, all array indexes used in the loop are i, which is the loop index. Therefore, it passes our legality check. Some loops may contain irregular memory accesses, e.g., `C[i] = A[B[i]]`. In this case, the index of array A depends on an element of array B. Static analysis cannot figure which element of array A is accessed in this statement. Thus, it is impossible for a compiler to directly divide the transfer of array A. We will describe how to regularize this type of accesses in order to enable *data streaming* in Section IV.

Memory allocation and deallocation. Usually, we allocate memory on the MIC when we need to copy data to the MIC. However, with *data streaming*, this will cause significant time overhead since the allocation procedure will be invoked many times. To avoid this, we do memory allocation only once before entering the loop. For each array, we allocate MIC memory for the entire array. For each scalar variable, its value is copied to the MIC at the allocation site. In the example of `blackscholes`, before entering the loop, we allocate MIC memory for the entire array $sptprice$ and copy values for variables $bsize$ and $price$ to the MIC. Similarly, all memory spaces are freed once after exiting the loop.

Loop transformation. To enable *data streaming*, the loop execution needs to be divided into blocks. To realize this, we replace the original loop with a two-level nested loop. The inner loop performs a computation block (i.e., a subset of continuous iterations of the original loop). The outer loop repeats until all computations are done. How to choose a proper block size will be described in the next section.

Data transfer and synchronization. Data transfer and synchronization primitives are inserted in the body of the outer loop to enable pipelined execution. In the i-th iteration of the outer loop, we first start the data transfer of the $(i+1)$-th block. We begin the execution of the i-th block on MIC only when the data transfer of the i-th block is done. Besides, the first data block is transferred before entering the loop.

B. Reducing Memory Usage

Offloading a loop to MIC requires properly using the MIC memory due to its limited size. MIC does not have disks attached to it. When offloaded data cannot be fit in the MIC memory, MIC will give out a runtime error. There is at most

8 GB memory available on MIC and part of it is reserved for OS. Applications with large memory footprints cannot be directly offloaded to MIC.

To solve this issue, we propose an optimization to reduce the memory usage of *data streaming*. When executing a loop in *data streaming* model, we in fact only need to reserve MIC memory for two data blocks – the data blocks for the current and the next computation blocks. None of the previous data blocks will be used anymore and their memory space can be reused. This not only reduces the memory usage on MIC, but also enables the offload of loops with larger input data.

Code transformation. Compared with the previous code transformation, there are two changes. Firstly, we only need to allocate MIC memory for two data blocks for each copied array. These memory blocks will be reused throughout the loop execution. Secondly, the outer loop contains two parts, one for odd blocks and the other for even blocks, as shown in sub-figure (c) of Figure 5. This is to transfer continuous data blocks into two different memory blocks on MIC. In the loop example, all even data blocks of array *sptprice* are stored in *sptprice1* and all odd data blocks are in *sptprice2*. The two memory blocks are reused for the entire array *sptprice*. Similarly, we create $prices\_b$ to hold data for the output array $prices$. Since we do not use asynchronous data transfer for the output, we only need one memory block for the output array.

Deciding the proper block size. Choosing a proper block size (i.e., number of iterations in one computation block) is critical to the loop performance on MIC. A larger block size will reduce the overhead of launching kernels, but increase the initial data transfer time. A smaller block size will save the initial data transfer time, but a lot of kernels might be launched.

Given a loop, we assume that the total data transfer time is D, the total computation time is C, the overhead of launching one kernel is K, and we split the loop into N blocks. Without *data streaming*, the total loop execution time is $D+K+C$. With *data streaming*, the total execution time can be calculated by using the following formula:

$$D/N + max\{C/N + K, D/N\} * (N - 1) + C/N + K$$

D/N is the data transfer time for the first block, $C/N + K$ is the computation time for the last block, and $max\{C/N + K, D/N\}$ is the execution time for any other block. When $C/N + K > D/N$, the best N value will be $sqrt(D/K)$. When $C/N + K \leq D/N$, the best N value will be $(D - C)/K$. In our experiments, we try N with value 10, 20, 40 and 50. We find that the best number of blocks for most benchmarks is between 10 and 40.

C. Minimizing Kernel Launches

The current LEO support for asynchronous data transfer and offload requires a kernel to be launched for each offload. The overhead of launching kernels can be significant if the same kernel is launched many times. We propose two optimizations to reduce this overhead.

Reusing MIC threads. Since the overhead of launching kernels may be high, we propose to reuse MIC threads in order to avoid repeated launches of the same kernels. The current Intel LEO does not provide support for reusing MIC threads. To reuse MIC threads, we launch only one loop kernel using asynchronous offload. While the kernel starts on MIC, CPU continues to transfer data blocks onto MIC in parallel. The kernel does not end after the computation block is done. Instead, it waits for the next data block. CPU will send MIC a signal when the next data block is ready. After receiving the signal, the kernel on MIC continues to compute the next block. The kernel exits only when the entire loop is done. In our implementation, we use lower-level COI library to control the synchronization between CPU and MIC.

```
+   pra  ma offload tar  et(mic:0)\
    for(i=0; i<iter; i++) {
        ...
-       pra  ma offload tar  et(mic:0)\
        ...
-       pra  ma offload tar  et(mic:0)\
        ...
-       pra  ma offload tar  et(mic:0)\
        ...
-       pra  ma offload tar  et(mic:0)\
        ...
-       pra  ma offload tar  et(mic:0)\
        ...
    }
```

Figure 6: Multiple offloads inside a `streamcluster` loop

```
    for (int i = 0 ; i < rows ; i++) {
        for (int j = 0; j < cols; j++) {
            k = i * cols + j;
            Jc = J[k];
            // irre  ular memory accesses
            dN[k] = J[iN[i] * cols + j] - Jc;
            dS[k] = J[iS[i] * cols + j] - Jc;
            dW[k] = J[i * cols + jW[j]] - Jc;
            dE[k] = J[i * cols + jE[j]] - Jc;
+       }
+   }
+   for (int i = 0 ; i < rows ; i++) {
+       for (int j = 0; j < cols; j++) {
+           k = i * cols + j;
            // the rest calculations
            G2 = (dN[k]*dN[k] + dS[k]*dS[k]
                + dW[k]*dW[k] + dE[k]*dE[k]) / (Jc*Jc);
            L = (dN[k] + dS[k] + dW[k] + dE[k]) / Jc;
            ...
        }
    }
```

Figure 7: Irregular memory accesses in benchmark `srad`

Merging offload. In many applications such as `streamcluster`, a large loop may contain multiple parallel inner loops. Each inner loop is offloaded, as shown in Figure 6. By applying *data streaming* to each individual

offload, we may cause significant overhead for launching kernels. To reduce the overhead, we merge the small offloads into a large offload and hoist the large offload out of the parent loop. In other words, instead of offloading those smaller inner loops, we offload the larger outer loop. The *in/out/inout* clauses of each inner loop are combined to populate the *in/out/inout* clauses for the outer loop. Although we may increase the sequential execution on MIC by doing this, we greatly reduce the kernel launching and data transfer overhead.

IV. Regularization

In real applications, parallel loops may contain irregular memory accesses. Figure 7 shows an example from benchmark `srad`. Each iteration of the inner loop reads array J and writes to array dN, dS, dW, and dE. The index of array J depends on the values of array iN, iS, jW, and jE. *Data streaming* cannot be directly applied here, since accesses to array J are not continuous and the mapping from array elements of J to iterations is unknown at compile time. This irregular access pattern also prevents many other compiler optimizations such as automatic vectorization from working properly. Vectorization is critical for MIC performance, since MIC provides 512-bit wide SIMD units. In addition, irregular accesses often increase data transfer time, because many elements in an array are transferred, but may not be accessed. Finally, irregular memory accesses may hurt cache performance, due to the lack of space locality. In this section, we propose to regularize irregular memory accesses in a parallel loop for better MIC performance.

We define an irregular memory access in a parallel loop as an access that does not access elements continuously across iterations. The regularization procedure transforms the access in order to make it access continuous elements across iterations. There are several common patterns of irregular accesses and we handle them separately.

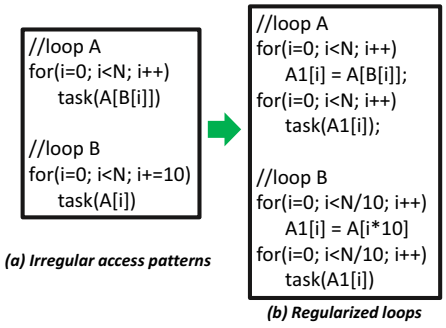

Figure 8: Two irregular memory access patterns

Reordering arrays. Figure 8 shows two common irregular access patterns in parallel loops. In the first loop, the index of array A is a value in array B. This disables *data streaming* and *vectorization*. In the second loop, the loop stride is a constant larger than 1, which is the case for benchmark `nn`. Since many elements are not used in the loop, transferring the entire array A will cause unnecessary data transfer and may also hurt cache performance. During regularization, for each of these two loops, we create a new array $A1$ which is a permutation of the original array A, as shown in Figure 8. The elements in the new array are sorted according to the access order in the original loop. By replacing the original array with the new one, we regularize all accesses inside the loop. The new array may contain repeated elements, because the same element of the original array may be accessed in more than one iteration. Since there is no cross-iteration dependence in parallel loops, repeated elements in the new array can only be read. If irregular memory accesses are write, values of the new array should be copied back to the original array after the loop. For safety, we apply the transformation only on arrays whose accesses are not guarded by any branch.

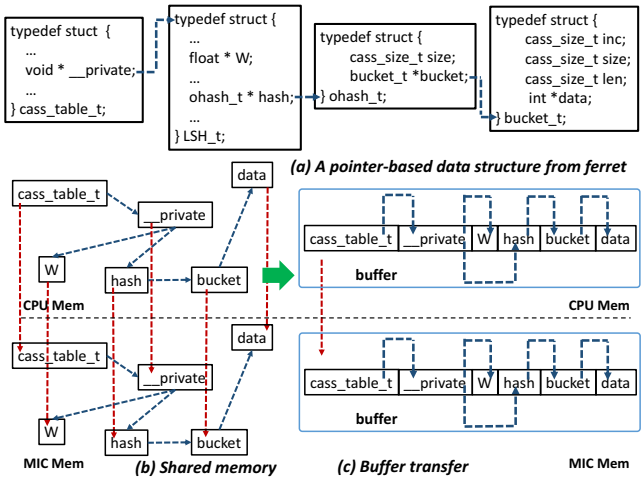

Figure 9: Data transfer for a pointer-based data structure from benchmark `ferret`. (a): a pointer-based data structure from `ferret`; (b): how MYO transfers the data structure; (c): how our method allocates memory for the data structure and transfers it to MIC.

Splitting loops. In real applications, we find that many loops perform irregular memory accesses at the beginning of each iteration. For example, as shown in Figure 7, after irregular accesses to array J, the remaining accesses are all regular. Since we only consider parallel loops (i.e., no cross-iteration dependences in the loops), the irregular memory accesses can be safely split from the rest of the loop body in this case. Figure 7 also shows the transformed `srad` loop after splitting, where we do all irregular accesses in the first loop and make the second loop regular. After splitting the loop, we can apply *vectorization* to the second loop.

Handling arrays of structures. Array of structures is another common irregular access pattern. Regularization can

be easily done by converting arrays of structures to structures of arrays statically.

Pipelining regularization with data transfer and computation. After regularization, *data streaming* can be applied to the loop. To save overhead, regularization can be done in parallel with data transfer and computation. More specifically, the regularization of block $i + 2$ can be done in parallel with the data transfer of block $i + 1$ and the computation of block i. The only extra overhead caused by regularization is the time taken to regularize the first data block.

V. Shared Memory for Large Pointer-based Data Structures

Intel MYO provides shared memory abstraction to support transferring complicated data structures such as pointer-based data structures between CPU and MIC. Sub-figure (a) in Figure 9 shows an example of a shared pointer-based data structure from benchmark `ferret`. The current MYO implements the virtual shared memory using a scheme similar to page fault handling. Shared data structures are copied on the fly at page level. When a shared variable is accessed on MIC, its entire page is copied to the MIC memory. This scheme is very slow due to several reasons: firstly, page granularity is too small for a large data structure; secondly, direct memory access (DMA) is not fully utilized for data transfer; finally, the large number of page faults may incur huge handling overhead. MYO only supports a limited number of shared memory allocations and a limited total size of shared memories. Applications with large memory footprinters cannot utilize the MYO scheme.

In our experiments, we observe that data transfer with larger granularity can greatly improve performance, when the loop is dealing with large pointer-based data structures. For example, copying data with 256 MB granularity can improve the performance of `ferret` by 7.81x. Based on this observation, we propose a technique to improve the performance of data transfer for large pointer-based data structures. As shown in Figure 9, our method preallocates large buffers for holding shared data structures. Data objects are created continuously in these preallocated buffers. When offloading a loop using these data structures, we copy entire data structures (i.e., entire preallocated buffers) to the MIC memory. When a shared object is accessed on MIC, our method does not need to check its state, since the entire object has been copied to the MIC memory. Therefore, the accesses to shared objects using our method are faster than MYO. Besides, our method makes full use of DMA during data transfer.

The next sections describe our methods to solve the two challenges: how the buffers are preallocated to minimize the memory usage on MIC, and how the links between objects are preserved after they are copied to MIC.

A. Buffer Allocation

A good strategy of buffer allocation should satisfy these two conditions: (1) the memory usage on MIC should be minimized, when the data structure is small, and (2) we should be able to fully utilize the entire memory space on MIC, when the data structure is large.

A straightforward way to allocate buffer is preallocating a very large buffer at the beginning. However, this may waste memory on MIC, when the data structure is small. Since MIC has limited memory and no disk attached, an underutilized large buffer may greatly affect the execution of other parts of the program or even other applications running on the same MIC. Another approach is to allocate a small buffer at first. Every time the buffer is full, we create a larger buffer and move the data into the new one. However, in this case, the buffer size is bounded by the largest continuous memory chunk OS can allocate, which is much smaller than the 8 GB memory size on MIC. According to the recent trend to larger data sets, many applications use data sets larger than 2 GB. In addition, this method may cause significant overhead for moving data.

We use a simple yet effective buffer allocation strategy that creates a set of separate buffers. More specifically, we create one buffer with a predefined size at the beginning. When the buffer is full, we create another one of the same size to hold new objects. In this way, we have one smaller buffer, when the data structure is small. When the data structure grows larger, we can fully utilize the entire memory space on MIC. In addition, we do not need to move data, when a new buffer is allocated.

B. Pointer Translation

After data structures are copied to the MIC memory, we need to preserve the links between objects to ensure the correctness of the loop execution. Pointer dereference on MIC is a challenging problem, since CPU and MIC have two separate memory spaces. The situation gets complicated by our discontinuous buffers.

Operations	**CPU**	**MIC**
*p	*(p.addr)	*(p.addr + delta[p.bid])
p1=p2	p1=p2	p1=p2
p=&obj	p.bid=obj.bid p.addr = &obj	p.bid=obj.bid p.addr = &obj - delta[p.bid]

Table I: Pointer operations on CPU and MIC

Since the program starts on a CPU, all pointers initially store CPU memory addresses. To make things simple, we restrict all shared pointers (i.e., pointers annotated with $\_Cilk\_shared$) to storing CPU memory addresses throughout the execution, even on MIC. When we dereference a shared pointer after the data structure is copied to MIC, we need to map a CPU memory address to the corresponding

Name	Source	Input	KLOC	Streaming	Merging	Regularization	Shared Memory
blackscholes	Parsec	10^7 options	0.415	✓(1.54)	-	-	-
streamcluster	Parsec	163840 points	1.79	✓(1.34)	✓(38.89)	-	-
ferret	Parsec	3500 images	11.159	-	-	-	✓(7.81)
dedup	Parsec	672 M data	2.319	-	-	-	-
freqmine	Parsec	250000 web docs	2.196	-	-	-	✓(1.16)
kmeans	Phoenix	100 clusters, 10^5 points	0.221	✓(1.95)	-	-	-
CG	NAS	75 K Array	0.524	✓(1.28)	✓(18.53)	-	-
cfd	Rodinia	53 M data	0.359	-	✓(27.19)	-	-
nn	Rodinia	$2.0 * 10^8$ points	0.12	✓(1.24)	-	✓(1.23)	-
srad	Rodinia	$4096 * 4096$ matrix	0.173	-	-	✓(1.25)	-
bfs	Rodinia	32 M points	0.138	-	-	-	-
hotspot	Rodinia	$1024 * 1024$ matrix	0.192	-	-	-	-

Table II: Benchmark information. From left to right: benchmark name, source of the benchmark, size of the input data, lines of code, and applicability of each optimization (speedup by individual optimization is given in the parentheses). Streaming — data streaming. Merging — offload merging.

MIC memory address. The pointer translation is done as follows:

When buffers containing pointer-based data structures are copied to the MIC memory, we create a table $delta$. The table size is the number of buffers we have copied to MIC. Each table entry corresponds to a pair of CPU (i.e., the source of the copy) and MIC (i.e., the destination of the copy) buffers. It contains the difference between two base addresses (i.e., the base address of the MIC buffer minus the base address of the CPU buffer).

When we perform translation for a pointer, we first identify which buffer the pointer points to. A straightforward method is to compare the pointer value with the base address of each buffer. However, this is very time-consuming, since it involves a set of comparison operations with the worst time complexity linear to the number of buffers. To fast locate the pointed buffer, we add a 1-byte field bid to each pointer and object annotated with $\_Cilk\_shared$. The bid field of a pointer stores the ID of the pointed buffer, while the bid field of an object stores the ID of the buffer it is located in. With this augmentation, we can directly get the buffer ID from a pointer's bid field, when it is being dereferenced. Table I summarized all pointer operations on CPU and MIC.

Finally, after we locate the buffer, we can get the MIC memory address from the CPU memory address by adding the home address difference (i.e., $delta[bid]$) to the pointer value.

VI. Evaluation

In this section, we show the experimental results of our optimizations.

We implement our optimizations as source-to-source code transformations using the Apricot framework [27]. Apricot provides modules for liveness analysis, handling of arrays, identification of offloadable code regions, and insertion of offload primitives. Most of our code transformations are done at AST level. We also use pycparser [9] for parsing codes into AST trees.

We begin our projects by porting various benchmarks to MIC. After identifying a performance bottleneck and designing an optimization, we look for other benchmarks that may benefit from the same optimization. In the end, we use 12 benchmarks from PARSEC [13], Phoenix [26], NAS [4], and Rodinia [17] benchmark suites in our experiment. All used benchmarks are summarized in Table II. Optimizations that can improve each benchmark are also shown in the table.

We compare the performance between a MIC and a multicore CPU in our experiment. The SKU of the MIC we used is ES2-P/A/X 1750. It has 61 cores at 1.05 GHz, 4 threads per each core, a total of 32 MB L2 cache and 8 GB GDDR5 memory. The CPU we used is Intel Xeon E5-2660, with 8 cores and 2.2 GHz clock frequency. All benchmarks are compiled using icc 14.0.1 with "-O2" optimization. The machine where we run our experiments is equipped with 67.6 GB memory, and the linux version we use is 2.6.32.

In the MIC versions of the benchmarks, we add pragmas to offload parallel loops to the MIC. We use 200 MIC threads for loops offloaded to the MIC. For most of the benchmarks, we use 4 CPU threads. Because the minimum thread numbers for dedup and ferret are 5 and 6, we use these two numbers of CPU threads when running them. For each version of each benchmark, we run it 10 times and record the execution time for the whole program. The variation among different runs is quite small. After normalization, the average variation is 1.30%, and the largest variation is 7.81%. We compare the average execution time to calculate speedups. The shortest and longest average execution times among all CPU versions are 10.06 seconds (dedup) and 320.49 seconds (ferret), respectively. The shortest and longest average execution times among all MIC optimized versions are 14.55 seconds (hotspot) and 232.43 seconds

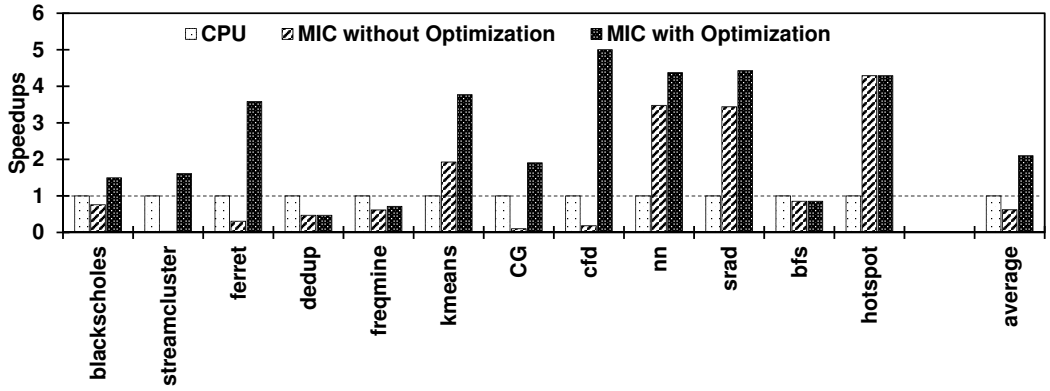

Figure 10: Application speedups over the original, parallel CPU implementation

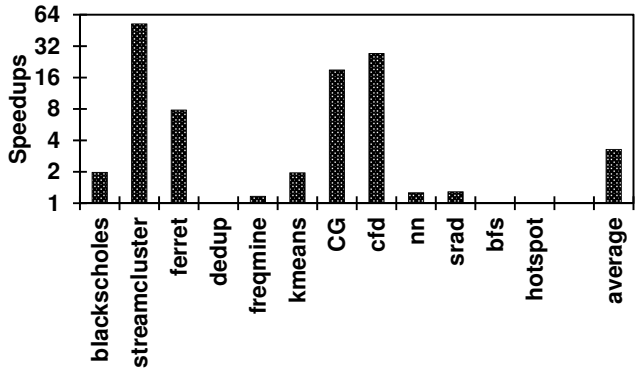

Figure 11: Application speedups achieved by our optimizations over the MIC versions w/o our optimizations

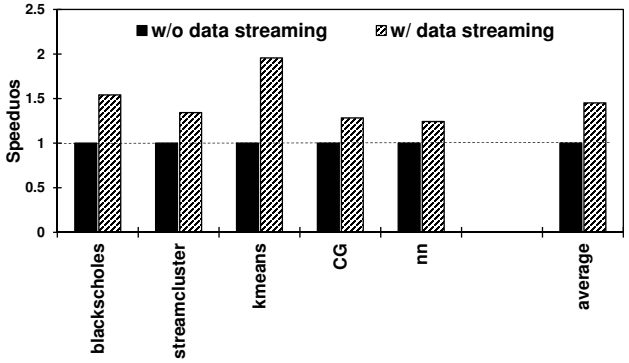

Figure 12: Performance gains by *data streaming*

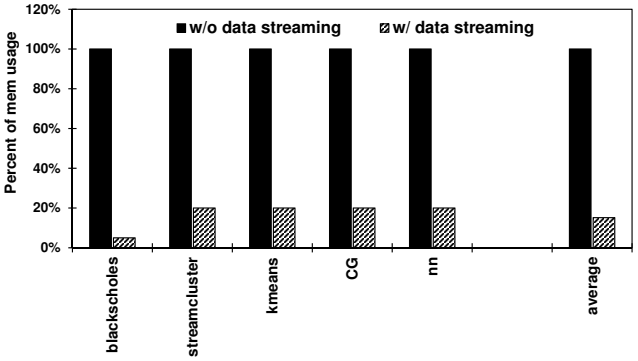

Figure 13: Memory usage after applying *data streaming*

(freqmine), respectively.

A. Overall Performance

Figure 10 compares the program performance with and without our optimizations. The speedups are measured for the entire benchmarks. All speedups are normalized to the speedups by the original multicore CPU versions. We can see that without our optimizations only 4 out of 12 benchmarks perform better on the MIC. Our optimizations make an additional 5 benchmarks achieve speedups on the MIC over their CPU versions. We can improve the performance of the applications by up to 5.0x speedups over their multicore CPU versions.

Figure 11 shows the relative speedups by our optimization over unoptimized MIC versions. As we can see, our optimizations improve the performance for 9 out of 12 benchmarks. For three benchmarks – `streamcluster`, `CG` and `cfd`, we achieve more than 16x speedups over the unoptimized versions. Benchmarks `bfs` and `hotspot` do not benefit from our optimizations, because their data transfer overheads are small compared to the computation time, and they do not contain any irregular memory access or use any shared memory. Benchmark `dedup` has *data streaming* implemented manually. Therefore, our optimizations do not bring any further speedup. Overall, we achieve from 1.16 to 52.21 times speedups compared to the MIC versions without our optimizations.

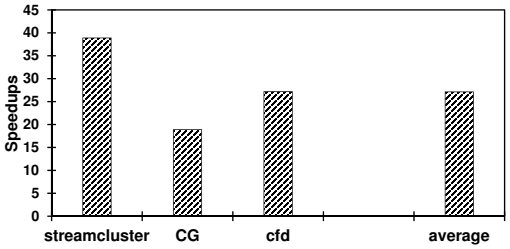

Figure 14: Performance gains by *offload merging*

B. Data Streaming

Data streaming benefits 5 out of 12 benchmarks we use. The speedups achieved by *data streaming* are shown in Figure 12. On average, *data streaming* achieves 1.45x speedup. Figure 13 shows the memory usage after applying *data streaming*. All numbers are normalized to the memory usage used by the original implementations on MIC. Because buffers for the transferred data are reused, *data streaming* greatly reduces the memory usage on MIC. For the benchmarks, *data streaming* reduces the memory usage by more than 80%. Figure 14 shows the speedups achieved by merging different offloads in the same loop. On average, merging offload regions can achieve 27.13x speedup.

C. Regularization

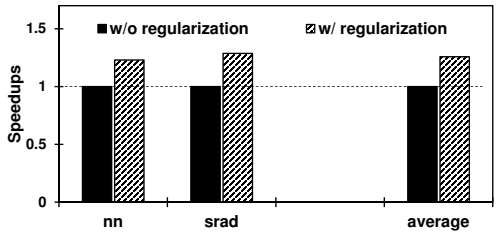

Figure 15: Performance gains by using *regularization*

Figure 15 shows the speedups achieved by using our *regularization* optimization. *Regularization* improves the performance of two benchmarks. For `nn` benchmark, we remove unnecessary data transfer. Around 2% execution time is spent on data copy. For `srad` benchmark, our optimization splits the loop containing irregular memory accesses into two, and vectorizes the part that does not contain irregular memory accesses. This optimization is done statically, and there is no runtime overhead. On average, *regularization* achieves 1.25x speedup.

D. Shared Memory

Table III summarizes the evaluation of our shared memory mechanism. There are two benchmarks that use large pointer-based data structures. We apply our shared memory mechanism to both of them. In our experiments, benchmark

Name	Static	Dynamic	Speedup
ferret	19	80298	7.81
freqmine	7	912	1.16

Table III: Performance gain by our shared memory mechanism. From left to right: benchmark name, number of shared allocations in the code, number of shared allocations performed at runtime, and speedups gained by our method.

`ferret` performs 80,298 shared memory allocations at runtime and the total usage of shared memory is 83 MB. It cannot run correctly using Intel MYO due to the large number of allocations. Our shared memory mechanism enables the use of MIC for `ferret`. The speedup in the table is measured over Intel MYO by using 1500 input images. Benchmark `freqmine` performs 912 shared memory allocations at runtime and requires 183 MB shared memory. Our shared memory mechanism can achieve 1.16x speedup.

VII. Related Work

Data streaming has been manually applied to individual GPU applications for improving performance, but none of the previous works propose a compiler optimization for automatically applying *data streaming*. Chakravarty et al. [15] use *data streaming* to overlap code generation with host-device data transfer in their code generator. Gregg and Hazelwood [19] have studied the impact of *data streaming* on a CUDA-based JPEG2000 encoder – CUJ2K [2]. Both Intel LEO [3] and NVIDIA CUDA [1] provide primitives to program asynchronous data transfer. Developers can use them to write GPU code with *data streaming*. But neither of them have the capability of automatically applying *data streaming* to applications.

Previous works have studied irregular memory accesses for GPU computing. Baskaran et al. [12] use a polyhedral compiler model to optimize affine memory references in regular loops. Yang et al. [29] use static analysis to identify memory access patterns and optimize the memory accesses through coalescing to achieve high data access bandwidth. Lee et al. [24] present a CUDA optimizer that uses a loop-collapsing technique to improve the performance of irregular applications. All these studies focus on the memory access patterns that are known at compile time. G-Streamline [30] is the work closest to *data reordering* in our *regularization* optimization. It also eliminates memory access irregularities at runtime. However, G-Streamline's goal is to make the data requested by a warp lie on a single memory segment to reduce the number of memory transactions on GPU, while our *regularization* optimization is to make the memory accesses continuous in order to enable *data streaming* and *vectorization*. Therefore, the transformed data accesses generated by G-Streamline are different from ours.

Virtual shared memory abstraction has been studied for CPU-GPU systems. DyMand [21], AMM [25] and

ADSM/GMAC [18] are runtime systems that implement runtime coherence mechanisms. They track read and write operations to monitor coherence status of data, similar to MYO [28]. Therefore, they suffer from the high overhead of tracking coherence states.

Prior works on data layout optimizations [31], [16], [22] have proposed to allocate memory together to increase spatial locality and improves cache performance on CPUs. Similar to our shared memory mechanism, they modify malloc sites and allocate one large chunk of memory instead of numerous small chunks for the array components distributed over memory space. Compared to their optimizations for CPUs, our shared memory mechanism is customized for manycore processors. Our memory allocation scheme can better utilize the limited memory on MIC. To handle pointers on MIC, we also propose an augmented design of pointers to fast translate pointers between their CPU and MIC memory addresses.

There are several works focusing on automatically offloading codes onto manycore coprocessors. Lee et al. [24] present a compiler that automatically converts OpenMP code to CUDA kernels. Apricot [27] automatically inserts LEO offload and data transfer clauses in OpenMP applications for MIC. The PGI [8] compiler automatically inserts data transfer clauses for OpenACC applications. None of these compilers handle the performance issues this paper targets.

VIII. Conclusions

This paper presents a set of compiler optimizations to improve the data transfer performance and memory utilization on Intel Xeon Phi coprocessors. These optimizations overlap data transfers with computations to hide the data transfer overhead, regularizes irregular memory accesses to enable data streaming and vectorization for irregular applications, and provides efficient support for data transfer and pointer translations for large pointer-based data structures. Although all the optimizations are presented in the context of Intel Xeon Phi coprocessors, we believe that these techniques can also be applied to other emerging manycore processors, such as the Tilera Tile-Gx processors. The experimental results show that our optimizations improve the performance of 9 out of 12 benchmarks. For 9 out of 12 benchmarks, we make their performance on a MIC better than that on a multicore CPU. Overall, our optimizations improve the MIC performance by 1.16x–52.21x.

References

[1] "CUDA," http://www.nvidia.com/object/cuda_home_new.html.

[2] "CUJ2K," http://cuj2k.sourceforge.net/.

[3] "Intel C++ Compiler," http://www.intel.com/Compilers.

[4] "NAS parallel benchmarks," https://www.nas.nasa.gov/publications/npb.html.

[5] "OpenACC: Directives for Accelerators," http://www.openacc-standard.org/.

[6] "OpenCL," https://developer.nvidia.com/opencl.

[7] "The OpenMP API," http://www.openmp.org.

[8] "The Portland Group (PGI)," http://www.pgroup.com.

[9] "pycparser," https://github.com/eliben/pycparser.

[10] "Threading Building Blocks," http://software.intel.com/en-us/articles/intel-tbb.

[11] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, ser. AFIPS '67 (Spring). New York, NY, USA: ACM, 1967, pp. 483–485.

[12] M. M. Baskaran, U. Bondhugula, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan, "A compiler framework for optimization of affine loop nests for GPGPUs," in *ICS*, 2008, pp. 225–234.

[13] C. Bienia, "Benchmarking modern multiprocessors," Ph.D. dissertation, Princeton University, January 2011.

[14] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An Efficient Multithreaded Runtime System," in *PPoPP*, 1995, pp. 207–216.

[15] M. M. Chakravarty, G. Keller, S. Lee, T. L. McDonell, and V. Grover, "Accelerating haskell array codes with multicore GPUs," in *DAMP*, 2011, pp. 3–14.

[16] S. Chatterjee, V. V. Jain, A. R. Lebeck, S. Mundhra, and M. Thottethodi, "Nonlinear Array Layouts for Hierarchical Memory Systems," in *ICS*, 1999.

[17] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *IISWC*, 2009, pp. 44–54.

[18] I. Gelado, J. E. Stone, J. Cabezas, S. Patel, N. Navarro, and W.-m. W. Hwu, "An Asymmetric Distributed Shared Memory Model for Heterogeneous Parallel Systems," in *ASPLOS*, 2010.

[19] C. Gregg and K. Hazelwood, "Where is the data? why you cannot debate CPU vs. GPU performance without the answer," in *ISPASS*, 2011, pp. 134–144.

[20] T. Hoshino, N. Maruyama, S. Matsuoka, and R. Takaki, "CUDA vs openACC: Performance case studies with kernel benchmarks and a memory-bound CFD application," in *CCGrid*, 2013, pp. 136–143.

[21] T. B. Jablin, J. A. Jablin, P. Prabhu, F. Liu, and D. I. August, "Dynamically Managed Data for CPU-GPU Architectures," in *CGO*, 2012.

[22] Y.-L. Ju and H. G. Dietz, "Reduction of Cache Coherence Overhead by Compiler Data Layout and Loop Transformation," in *LCPC*, 1992.

[23] S. Lee and R. Eigenmann, “OpenMPC: Extended OpenMP Programming and Tuning for GPUs,” in *SC*, 2010.

[24] S. Lee, S.-J. Min, and R. Eigenmann, “OpenMP to GPGPU: a compiler framework for automatic translation and optimization,” in *PPoPP*, 2009, pp. 101–110.

[25] S. Pai, R. Govindarajan, and M. J. Thazhuthaveetil, “Fast and Efficient Automatic Memory Management for GPUs Using Compiler-Assisted Runtime Coherence Scheme,” in *PACT*, 2012.

[26] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis, “Evaluating mapreduce for multi-core and multiprocessor systems,” in *HPCA*, 2007, pp. 13–24.

[27] N. Ravi, Y. Yang, T. Bao, and S. Chakradhar, “Apricot: an Optimizing Compiler and Productivity Tool for x86-Compatible Many-Core Coprocessors,” in *ICS*, 2012, pp. 47–58.

[28] B. Saha, X. Zhou, H. Chen, Y. Gao, S. Yan, M. Rajagopalan, J. Fang, P. Zhang, R. Ronen, and A. Mendelson, “Programming Model for a Heterogeneous x86 Platform,” in *PLDI*, 2009.

[29] Y. Yang, P. Xiang, J. Kong, and H. Zhou, “A GPGPU compiler for memory optimization and parallelism management,” in *PLDI*, 2010, pp. 86–97.

[30] E. Z. Zhang, Y. Jiang, Z. Guo, K. Tian, and X. Shen, “On-the-fly elimination of dynamic irregularities for GPU computing,” in *ASPLOS*, 2011, pp. 369–380.

[31] Y. Zhang, W. Ding, J. Liu, and M. Kandemir, “Optimizing Data Layouts for Parallel Computation on Multicores,” in *PACT*, 2011.